System Programming Project 3

담당 교수 : 박성용 교수님

이름 : 김규빈

학번 : 20200152

1. 개발 목표

- 해당 프로젝트에서 구현할 내용을 간략히 서술.
- (주식 서버를 만드는 전체적인 개요에 대해서 작성하면 됨.)

다수의 고객이 동시에 접속하여 작업을 수행할 수 있는 동시성을 가진 주식 서버를 구현하는 것을 목표로 한다. 고객은 서버에 주식 조회, 구매, 판매를 요청할 수 있으며, 서버는 이러한 요청을 받아 처리한다. 여러 고객이 동시에 서버에 요청을 보낼 수 있으므로, 이를 효율적으로 처리하기 위해 다음 두 가지 방식을 사용하여 주식 서버를 구현한다:

Event Driven Approach (이벤트 기반 접근 방식)

Thread Based Approach (스레드 기반 접근 방식)

이 두 가지 접근 방식을 통해 서버의 성능과 확장성을 극대화하고, 다양한 상황에서의 동시성 문제를 해결하고자 한다.

2. 개발 범위 및 내용

A. 개발 범위

Event Driven Approach와 Thread Based Approach 모두 서버 시작 시 stock.txt 파일을 읽어 이진 트리 형식으로 데이터를 저장한다. 서버 종료 시 변경된 정보를 다시 stock.txt 파일에 덮어쓴다.

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach
    - 설명 : Event Driven Approach에서는 하나의 스레드 내에서 모든 고객의 요청을 처리한다.
    - 작동 방식 : 'select' 함수를 사용하여 여러 파일 디스크립터를 관리하고, 요청이 발생하면 해당 요청을 처리한 후 다시 'select' 함수를 통해 다음 요청을 대기하는 방식으로 동작한다.
    - 특징 : 이 접근 방식은 다수의 요청이 동시에 들어와도 이를 절차적으로 수행하여 서버가 동시성을 가지도록 한다.

2. Task 2: Thread-based Approach

   - 설명 : Thread Based Approach에서는 각 client의 요청을 별도의 스레드에서 처리한다.
   - 작동 방식 : 오버헤드를 최소화하기 위해 미리 여러 개의 스레드를 생성해 두고, client의 연결 요청이 들어오면 사용 중이지 않은 스레드를 해당 고객과 연결하여 동작한다.
   - 특징 : Event Driven Approach와 달리, 여러 스레드에서 작업이 동시적으로 수행되므로 multicore를 사용할 때 이점을 얻을 수 있다. 또한, 스레드 간에 공유하는 데이터를 관리하기 위해 'semaphore'를 사용한다.

3. Task 3: Performance Evaluation

   - 설명 : 'multiclient.c'를 통해 확장성과 workload에 따른 분석을 수행한다.
   - 작동 방식 : 원활한 측정을 위해 'usleep'을 주석 처리하거나 워크로드 조건에 따라 특정 코드를 추가하는 등 'multiclient.c' 파일을 수정한다.
   - 평가 방법 : 각 경우마다 세번의 실험을 실시하고, 그 평균으로 성능을 측정한다.

B. 개발 내용

- **아래 항목의 내용만 서술**
- **(기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)**
- **Task1 (Event-driven Approach with select())**

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

  - I/O Multiplexing 구현: 고객과 정보를 주고받을 파일 디스크립터들이 저장된 fd_set과 select 함수를 통해 I/O Multiplexing을 구현한다.
  - 다수의 파일 디스크립터 관리: select 함수를 사용하여 여러 파일 디스크립터를 관리한다. select 함수는 인자로 받은 fd_set 내의 이벤트가 발생한 파일 디스크립터들의 fd_set을 반환한다.
  - 새로운 연결 처리: 반환받은 fd_set 내의 이벤트가 발생한 파일 디스크립

터가 listenfd인 경우, 새로운 고객의 연결 요청이 들어온 것이므로 이를 fd_set에 저장한다.

- ◆ 기존 연결 처리: 이벤트가 발생한 파일 디스크립터가 listenfd가 아닌 경우, 기존에 연결된 고객의 요청이 들어온 것이므로 해당 요청에 맞게 이를 처리해준다.
- ◆ 다수의 요청 처리: 다수의 요청이 발생한 경우, 이벤트가 발생한 파일 디스크립터들을 순서대로 순회하면서 요청을 처리하여 서버가 동시성을 가지도록 한다.

✓ epoll과의 차이점 서술

- ◆ 효율적인 I/O Multiplexing: epoll은 select의 단점을 보완한 함수로, 보다 효율적인 I/O Multiplexing을 구현할 수 있도록 도와준다.
- ◆ 향상된 성능: epoll은 이벤트가 발생한 소켓만을 처리하기 때문에 select보다 향상된 성능을 보장하며, 소켓 수에 관계없이 일정한 성능을 유지한다.
- ◆ epoll은 이벤트가 발생할 때까지 블로킹되지 않아, 보다 효율적으로 프로그램을 구현할 수 있게 도와준다.

- **Task2 (Thread-based Approach with pthread)**

✓ Master Thread의 Connection 관리

- ◆ Connection 관리 개요: 오버헤드를 줄이기 위해 미리 스레드를 생성하고, 새로운 고객의 요청이 발생할 때마다 스레드를 연결한다. 자세한 과정은 다음과 같다.
- ◆ sbuf 배열 사용: 아직 스레드에서 작업 중이지 않은 파일 디스크립터를 저장할 배열인 sbuf 배열에 새로운 고객에 대해 생성된 파일 디스크립터를 계속해서 저장한다. 동시에 대기 중인 스레드는 sbuf 내에 스레드가 배정받지 않은 파일 디스크립터가 존재하는지 확인한다.
- ◆ 파일 디스크립터 처리: 만약 그러한 파일 디스크립터가 존재할 경우 이를 추출하여 해당 파일 디스크립터에 대한 작업을 수행하게 된다. 결과적으로 하나의 스레드가 특정 고객의 요청에 대해 독립적으로 작업을 수행하게 된다.

◆ 문제점 : 이 과정에서 예상치 못한 컨텍스트 스위칭이나 여러 스레드에서 동일한 파일 디스크립터를 추출해가는 문제가 발생할 수 있다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

◆ Semaphore 사용 : 위에서 언급한 문제를 해결하기 위해 semaphore를 사용한다.

◆ Semaphore 변수 : sbuf 내에 존재하는 파일 디스크립터의 개수를 나타내는 items, 그리고 sbuf에 파일 디스크립터를 저장할 수 있는 공간의 개수를 나타내는 slots라는 세마포어 변수를 통해 sbuf의 공간을 관리한다. 또한, mutex라는 세마포어 변수를 통해 두 스레드가 sbuf에 동시에 접근하여 하나의 파일 디스크립터를 공유하는 문제를 방지한다.

◆ 주식 노드 접근 관리 : 파일 디스크립터 외에도 여러 스레드가 동시에 접근하여 그 값을 조회 및 수정하는 각 주식 노드 역시 세마포어 변수를 통해 관리해야 한다. 노드를 조회하는 show 명령어의 경우 여러 스레드가 동시에 접근할 수 있지만, 노드를 수정하는 buy, sell 명령어의 경우 하나의 스레드만 접근할 수 있으므로 이 두 경우를 분리해야 한다.

◆ Semaphore와 변수 관리 : 각 노드에 대해 그 노드에 접근하고 있는 스레드의 수를 나타내는 readcnt 변수와 readcnt 관리를 위한 mutex 세마포어 변수, 마지막으로 현재 노드 접근이 조회인지 수정을 나타내는 w 세마포어 변수를 통해 이를 관리한다.

- **Task3 (Performance Evaluation)**

✓ 얻고자 하는 metric 정의, 그렇게 정한 이유, 측정 방법 서술

◆ 확장성 비교 : client 수에 따른 Event driven Approach와 Thread Based Approach의 동시 처리율을 비교한다. client 수가 1, 10, 50, 100인 경우에 대해 실험을 진행한다.

◆ Workload 분석 : 클라이언트 측에서 show, buy, sell을 모두 요청하는 경우, buy와 sell만 요청하는 경우, show만 요청하는 경우의 세 가지 시나리오에 대해 실험을 진행한다. 이 경우에도 클라이언트 수가 1, 10, 50, 100인 경우에 대해 실험을 진행한다.

◆ 측정 방법 : 각 실험의 결과는 3번의 실행 평균값을 이용하며, 시간은

'gettimeofday' 함수를 사용하여 초 단위로 측정한다.

- ✓ Configuration 변화에 따른 예상 결과 서술
  - ◆ 동시 처리율 : Event-driven Approach와 Thread-Based Approach 모두 클라이언트 수가 증가함에 따라 동시 처리율이 증가할 것으로 예상된다. 이는 두 접근 방식 모두 클라이언트의 동시 요청을 처리할 수 있는 구조를 가지고 있기 때문이다. 하지만 두 접근 방식의 오버헤드 차이로 인해 동시 처리율의 증가율은 다를 수 있다.
    - Event-driven Approach
      - 이 방식은 하나의 스레드에서 모든 작업을 절차적으로 처리하기 때문에, 많은 수의 클라이언트 요청이 있을 때 오버헤드가 증가할 수 있다.
      - 클라이언트 수가 증가할수록 이벤트 loop의 부담이 커져서 상대적으로 동시 처리율이 낮을 것으로 예상된다
    - Thread-Based Approach
      - 이 방식은 여러 스레드에서 작업을 병렬로 처리하므로 멀티코어 환경의 이점을 최대한 활용할 수 있다.
      - 클라이언트 수가 증가할수록 더 많은 스레드를 활용할 수 있어 동시 처리율이 더욱 높아질 것으로 예상된다.

    따라서, 클라이언트 수가 많아질수록 Thread-Based Approach의 동시 처리율이 Event-driven Approach보다 더 높을 것으로 예상된다.
  - ◆ Workload에 따른 분석 :
    - 모든 클라이언트가 buy 또는 sell을 요청하는 경우
      - 동시 처리율: buy와 sell 명령어는 주식 데이터를 수정해야 하기 때문에, 이들 명령어만 요청하는 경우 동기화 오버헤드가 발생할 수 있다. 특히, Thread-Based Approach에서는 각 스레드가 주식 데이터를 병렬로 수정하려고 할 때 발생하는 동기화 문제로 인해 성능이 저하될 수 있다. 따라서 동시 처리율이 모든 명령어를 섞어서 요청할 때보다 낮을 수 있다.

- Event-driven Approach: 이 접근 방식에서는 하나의 스레드가 모든 작업을 처리하기 때문에, 동시 처리율이 상대적으로 낮을 것으로 예상된다. 클라이언트 수가 증가함에 따라 동기화 오버헤드가 누적되어 성능이 저하될 가능성이 크다.

- Thread-Based Approach: 여러 스레드가 병렬로 작업을 처리하므로, 멀티코어 환경에서 더 높은 동시 처리율을 기대할 수 있다. 하지만 동기화 오버헤드 때문에 예상보다 높은 성능을 보이지 않을 수도 있다. 그래도 Event-driven Approach보다 높은 동시 처리율을 보일 것으로 예상된다.

- 모든 클라이언트가 show만 요청하는 경우

  - 동시 처리율: show 명령어는 주식 데이터를 읽기만 하므로, 동기화 오버헤드가 발생하지 않는다. 따라서, show 명령어만 요청할 때 가장 높은 동시 처리율을 기대할 수 있다.

  - Event-driven Approach: 하나의 스레드가 모든 요청을 처리하지만, 읽기 작업만 수행하므로 동기화 오버헤드가 없고, 비교적 높은 동시 처리율을 유지할 수 있다. 클라이언트 수가 증가함에 따라 동시 처리율이 증가할 것으로 예상된다.

  - Thread-Based Approach: 각 스레드가 병렬로 읽기 작업을 수행하므로, 멀티코어 환경에서 매우 높은 동시 처리율을 보일 수 있다. 모든 클라이언트가 show 명령어만 요청하는 경우, Event-driven Approach보다 높은 동시 처리율을 보일 것으로 예상된다.

- 클라이언트가 buy, show 등을 섞어서 요청하는 경우

  - 동시 처리율: 다양한 명령어가 혼합되어 요청될 때, 각 명령어의 비용과 명령어 간의 상호작용이 성능에 영향을 미친다. 따라서, 동시 처리율은 클라이언트가 특정 명령어만 요청할 때보다는 낮을 수 있다.

  - Event-driven Approach: 하나의 스레드에서 모든 요청을 처리하므로, 혼합된 명령어를 처리할 때 발생하는 오버헤드로 인해 동시 처리율이 낮아질 수 있다. 특히, show 명령어와 같은 높은 비용의 명령어가 포함될 때 성능 저하가 발생할 수 있다.

- Thread-Based Approach: 각 명령어가 별도의 스레드에서 병렬로 처리되므로, 전체적인 성능이 Event-driven Approach보다 우수할 것으로 예상된다. 하지만, show 명령어와 같은 높은 비용의 명령어가 포함되면 성능 저하가 발생할 수 있다. 그럼에도 불구하고, Thread-Based Approach는 멀티코어 환경의 이점을 잘 활용하여 상대적으로 높은 동시 처리율을 유지할 것으로 예상된다.

C. 개발 방법

- **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

- Event-Driven Approach

1. Stock 구조체

```
typedef struct Stock
{
    int id;
    int left_stock;
    int price;
    struct Stock *left;
    struct Stock *right;
    pthread_mutex_t lock;
} Stock;
```

주식 정보를 노드 단위로 관리하기 위한 구조체이다. 주식의 ID, 남은 주식 수, 주식의 가격을 담는다. 전체 주식의 이진 트리 저장을 위해 left, right 포인터 변수가 존재한다. Thread-Based Approach 구현 시 thread의 독립적인 접근을 위한 pthread_mutex_t 변수도 존재한다.

A. Stock 관련 함수

```
void show_stock(int connfd);
```

고객으로부터 "show" 명령을 받았을 때 실행하는 함수이다. 이진 트리를 순회하며 주식의 id, 남은 상품 수, 가격을 담은 문자열을 생성한 뒤 다시 고객에게 전송한다. Thread-Based Approach의 경우 노드 단위로 발생할 수 있는 Reader-Writer Problem을 해결해야 한다. 특정 노드에 접근을 시작하면 mutex를 이용해 다른 접근을 막은 뒤 readcnt를 증가시킨다. 첫 번째 접근이었다면 w semaphore 변수를 0으로 감소시킨다. 만약 다른 thread가 해당 노드에 대해 수정을 수행 중이었다면 해당 과정에서 수정이 완료될 때까지 기다리게 된다. 이후 해당 노드에 대한 정보를 문

자열에 담은 뒤 해당 노드에 대한 조회가 완료됐으므로 다시 mutex를 이용해 다른 접근을 막은 뒤 readcnt를 감소시킨다. 역시 마찬가지로 해당 thread가 마지막 접근이었다면 w semaphore 변수를 1로 증가시켜 다시 다른 thread의 수정 접근이 가능하게끔 만들어준다.

```
void buy_stock(int connfd, int target_id, int quantity);
void sell_stock(int connfd, int target_id, int quantity);
```

고객으로부터 "buy" 혹은 "sell" 명령을 받았을 때 실행하는 함수이다. 이진 트리를 순회하며 사거나 팔 노드를 검색하고 주어진 수만큼 해당 주식을 증가시키거나 감소시킨다. 성공하거나 실패할 수 있으므로 그에 해당하는 메시지를 다시 고객에게 전송한다. Thread-Based Approach의 경우 노드 단위로 발생할 수 있는 Reader-Writer Problem을 해결해야 한다. 수정이 필요한 특정 노드에 접근을 시작하면 w semaphore 변수를 감소시켜 해당 노드의 조회나 수정을 막는다. 만약 다른 thread에서 해당 노드를 조회하거나 수정 중이라면 완료될 때까지 기다리게 된다.

2. ClientPool 구조체

```
typedef struct
{
    int maxfd;        // 가장 큰 디스크립터 (select 함수의 첫 번째 매개변수)
    fd_set read_set;  // 모든 활성 디스크립터 집합
    fd_set ready_set; // 읽기 가능한 디스크립터 집합
    int nready;       // 준비된 디스크립터 수
    int maxi;         // clientfd의 가장 큰 인덱스
    int clientfd[FD_SETSIZE];
    rio_t clientrio[FD_SETSIZE];
} ClientPool;
```

Event-Driven Approach를 구현하기 위해 관리 중인 file descriptor를 저장하는 구조체이다. 관리 중인 file descriptor를 담는 read_set과 select 함수의 반환 값을 저장할 ready_set이 존재하며, 고객들과 정보를 주고받기 위한 rio 구조체를 담을 배열이 존재한다.

A. ClientPool 구조체 관련 함수

```
void add_client(int clientfd, ClientPool *p);
```

select 함수를 통해 이벤트 발생이 확인된 file descriptor 중 listenfd가 존

재할 시에 수행되는 함수이다. 새로운 고객의 연결 요청이 발생했으므로 accept를 통해 얻은 file descriptor를 read_set에 추가한다. 또한, 해당 file descriptor에 대해 사용할 rio 구조체의 생성 및 pool 내의 관련 변수의 수정이 발생한다.

```
void check_client(ClientPool *p);
```

select 함수를 통해 이벤트 발생이 확인된 file descriptor 중 listenfd를 제외한 file descriptor에 대한 처리를 수행한다. 이벤트 발생이 확인된 file descriptor에 대해 받아진 명령에 따라 show_stock, buy_stock, sell_stock 함수를 시행한다.

- Thread-based Approach

1. sbuf 구조체

```
typedef struct
{
    int *buf;     // 버퍼 포인터
    int n;        // 버퍼 크기
    int front;    // 버퍼 앞쪽 인덱스
    int rear;     // 버퍼 뒷쪽 인덱스
    sem_t mutex; // mutex semaphore
    sem_t slots; // 슬롯 semaphore
    sem_t items; // 아이템 semaphore
} sbuf_t;
```

고객의 연결로 생성된 file descriptor를 보관하고 하나의 file descriptor에 대해 복수의 thread에서의 접근을 막기 위한 구조체이다. file descriptor를 저장할 buf 변수, 총 저장 가능 개수, 처음과 마지막 file descriptor의 위치, 그리고 semaphore 변수들을 포함한다.

A. sbuf 구조체 관련 함수

```
void sbuf_insert(sbuf_t *sp, int connfd);
```

file descriptor를 sbuf에 저장한다. 이때 slots semaphore 변수를 감소시키며, 만약 남는 slot이 존재하지 않는다면 sbuf_remove에서 slots를 증가시켜줄 때까지 기다리게 된다. 이후에 mutex로 동시 접근을 막고 buf에 file descriptor를 추가하며 item semaphore 변수를 증가시킨다.

```
int sbuf_remove(sbuf_t *sp);
```

file descriptor를 sbuf에서 제거한다. 이때 items semaphore 변수를 감소시키며, 만약 남는 item이 존재하지 않는다면 sbuf_insert에서 items를 증가시켜줄 때까지 기다리게 된다. 이후에 mutex로 동시 접근을 막고 buf에서 file descriptor를 제거하며 slots semaphore 변수를 증가시킨다.

2. Stock 구조체

```
typedef struct Stock
{
    int id;              // 재고 ID
    int left_stock;      // 남은 재고 수량
    int price;           // 가격
    struct Stock *left;  // 왼쪽 자식 노드 포인터
    struct Stock *right; // 오른쪽 자식 노드 포인터
    int readcnt;         // 읽기 카운트
    sem_t mutex;         // mutex semaphore
    sem_t w;             // 쓰기 semaphore
} Stock;
```

주식 정보를 노드 단위로 관리하기 위한 구조체이다. 주식의 ID, 남은 주식 수, 주식의 가격을 담는다. 전체 주식의 이진 트리 저장을 위해 left, right 포인터 변수가 존재한다. Thread-Based Approach 구현 시 thread의 독립적인 접근을 위한 semaphore 변수와 readcnt 변수도 존재한다.

A. Stock 관련 함수

```
Stock *find_stock(int id);
```

고객으로부터 "show" 명령을 받았을 때 실행하는 함수이다. 이진 트리를

순회하며 주식의 id, 남은 상품 수, 가격을 담은 문자열을 생성한 뒤 다시 고객에게 전송한다. Thread-Based Approach의 경우 노드 단위로 발생할 수 있는 Reader-Writer Problem을 해결해야 한다. 특정 노드에 접근을 시작하면 mutex를 이용해 다른 접근을 막은 뒤 readcnt를 증가시킨다. 첫 번째 접근이었다면 w semaphore 변수를 0으로 감소시킨다. 만약 다른 thread가 해당 노드에 대해 수정을 수행 중이었다면 해당 과정에서 수정이 완료될 때까지 기다리게 된다. 이후 해당 노드에 대한 정보를 문자열에 담은 뒤 해당 노드에 대한 조회가 완료됐으므로 다시 mutex를 이용해 다른 접근을 막은 뒤 readcnt를 감소시킨다. 역시 마찬가지로 해당 thread가 마지막 접근이었다면 w semaphore 변수를 1로 증가시켜 다시 다른 thread의 수정 접근이 가능하게끔 만들어준다.

```
void buy_stock(int connfd, int target_id, int quantity);
void sell_stock(int connfd, int target_id, int quantity);
```

고객으로부터 "buy" 혹은 "sell" 명령을 받았을 때 실행하는 함수이다. 이진 트리를 순회하며 사거나 팔 노드를 검색하고 주어진 수만큼 해당 주식을 증가시키거나 감소시킨다. 성공하거나 실패할 수 있으므로 그에 해당하는 메시지를 다시 고객에게 전송한다. Thread-Based Approach의 경우 노드 단위로 발생할 수 있는 Reader-Writer Problem을 해결해야 한다. 수정이 필요한 특정 노드에 접근을 시작하면 w semaphore 변수를 감소시켜 해당 노드의 조회나 수정을 막는다. 만약 다른 thread에서 해당 노드를 조회하거나 수정 중이라면 완료될 때까지 기다리게 된다.

3. Thread

```
void *thread(void *vargp);
```

생성된 thread에서 시행될 함수이다. Pthread_detach(pthread_self)를 통해 thread 함수의 종료와 함께 thread가 자동으로 종료된다. sbuf 배열에서 file descriptor를 하나 추출해 해당 file descriptor를 통해 들어오는 요청에 대해 받아진 명령에 따라 show_stock, buy_stock, sell_stock 함수를 시행한다.

3. 구현 결과

- **2번의 구현 결과를 간략하게 작성**

- 미처 구현하지 못한 부분에 대해선 디자인에 대한 내용도 추가

✓ **Task1(Event-driven Approach)**

◆ Task1은 select 함수를 사용하여 이벤트 기반으로 클라이언트의 요청을 처리한다. ClientPool 구조체를 통해 모든 file descriptor를 관리하며, select 함수는 발생한 이벤트를 감지하고, 이벤트가 발생한 file descriptor를 처리한다.

- ClientPool 구조체: 모든 활성 디스크립터 집합과 준비된 디스크립터 집합을 관리한다.
- init_pool: ClientPool 구조체를 초기화한다.
- add_client: 새로운 클라이언트가 연결될 때 ClientPool에 추가한다.
- check_client: 이벤트가 발생한 클라이언트를 확인하고 요청을 처리한다.
- 명령 처리: 클라이언트의 명령어(show, buy, sell, exit)를 파싱하고 처리한다.

◆ 이벤트가 발생한 file descriptor가 listenfd일 경우, add_client 함수를 통해 ClientPool에 새로운 file descriptor를 추가하고, 그 외에서 이벤트가 발생할 경우 check_clients를 통해 관리하는 모든 file descriptor를 순회하며 동시적인 요청을 절차적으로 처리한다.

✓ **Task2(Thread-based Approach)**

◆ Task2는 각 클라이언트 연결에 대해 새로운 스레드를 생성하여 클라이언트의 요청을 처리한다. 이는 context switching을 통해 동시성을 제공한다.

- sbuf_t 구조체: 공유 버퍼 구조체로, 스레드 간의 작업을 관리한다.
- sbuf_init: 공유 버퍼를 초기화한다.

- sbuf_insert: 공유 버퍼에 연결된 클라이언트를 삽입한다.
- sbuf_remove: 공유 버퍼에서 클라이언트를 제거한다.
- thread 함수: 각 스레드가 실행될 함수로, 클라이언트 요청을 처리한다.
- handle_client: 클라이언트의 요청을 처리한다.

◆ 새로운 클라이언트의 연결로 인해 file descriptor가 생성될 때마다 하나의 스레드를 배정하여, 해당 스레드에서 특정 file descriptor를 통해 들어오는 요청을 처리한다. 이 때 스레드 간의 공유 자원에 대한 동기화 처리가 필요하다.

✓ **Task3**

◆ multiclient.c 파일의 값들을 수정하여 고객의 수 및 고객이 요청하는 명령어의 종류를 조정할 수 있다. 이를 통해 다양한 시나리오에서 서버의 성능과 동작을 테스트할 수 있다.

4. 성능 평가 결과 (Task 3)

- **강의자료 슬라이드의 내용 참고하여 작성 (측정 시점, 출력 결과 값 캡처 포함)**

✓ 확장성 : Event-driven Approach와 Thread-Based Approach 모두 클라이언트 수가 증가함에 따라 동시 처리율이 증가하는 것을 확인할 수 있다. 그러나 두 접근 방식의 오버헤드 차이로 인해 동시 처리율의 증가율은 다르다.

- Event-driven Approach – 1 client
- `Total elapsed time for all clients: 0.059565 seconds`
- `Total elapsed time for all clients: 0.054455 seconds`
- `Total elapsed time for all clients: 0.066790 seconds`
- Event-driven Approach – 10 clients
- `Total elapsed time for all clients: 0.599305 seconds`
- `Total elapsed time for all clients: 0.560385 seconds`
- `Total elapsed time for all clients: 0.622694 seconds`

- Event-driven Approach – 50 clients
  - `Total elapsed time for all clients: 3.439288 seconds`
  - `Total elapsed time for all clients: 2.511449 seconds`
  - `Total elapsed time for all clients: 2.459569 seconds`
- Event-driven Approach – 100 clients
  - `Total elapsed time for all clients: 5.340226 seconds`
  - `Total elapsed time for all clients: 5.015325 seconds`
  - `Total elapsed time for all clients: 5.875161 seconds`
- Thread-based Approach – 1 client
  - `Total elapsed time for all clients: 0.048480 seconds`
  - `Total elapsed time for all clients: 0.052351 seconds`
  - `Total elapsed time for all clients: 0.079414 seconds`
- Thread-based Approach – 10 clients
  - `Total elapsed time for all clients: 0.542926 seconds`
  - `Total elapsed time for all clients: 0.653285 seconds`
  - `Total elapsed time for all clients: 0.522852 seconds`
- Thread-based Approach – 50 clients
  - `Total elapsed time for all clients: 2.854542 seconds`
  - `Total elapsed time for all clients: 2.510160 seconds`
  - `Total elapsed time for all clients: 2.598230 seconds`
- Thread-based Approach – 100 clients
  - `Total elapsed time for all clients: 5.123416 seconds`
  - `Total elapsed time for all clients: 5.036947 seconds`
  - `Total elapsed time for all clients: 5.203286 seconds`

◆ Event-driven Approach:

- 클라이언트 수가 1일 때 동시 처리율은 165.92, 10일 때는 168.31, 50

일 때는 178.35, 100일 때는 184.83로 증가한다.

- 평균 응답 시간은 1명의 클라이언트가 요청할 때 0.06027초에서 100명의 클라이언트가 요청할 때 5.410237초로 증가했다.
- 동시 처리율이 클라이언트 수가 증가할수록 증가하지만, 증가율은 클라이언트 수가 많아질수록 감소하는 경향을 보였다. 이는 하나의 스레드에서 모든 작업을 처리하기 때문에 발생하는 오버헤드 때문이다.

◆ Thread-Based Approach:

- 클라이언트 수가 1일 때 동시 처리율은 166.44, 10일 때는 174.51, 50일 때는 188.37, 100일 때는 195.27로 증가한다.
- 평균 응답 시간은 1명의 클라이언트가 요청할 때 0.0600817초에서 100명의 클라이언트가 요청할 때 5.1212163초로 증가했다.
- 동시 처리율이 클라이언트 수가 증가할수록 증가하며, Event-driven Approach보다 더 높은 증가율을 보였다. 이는 여러 스레드가 병렬로 작업을 처리하므로 멀티코어 환경의 이점을 최대한 활용할 수 있기 때문이다.

◆ Thread-Based Approach는 멀티코어 환경에서 여러 스레드를 활용하여 작업을 병렬로 처리할 수 있기 때문에, 클라이언트 수가 증가함에 따라 동시 처리율이 더 크게 증가하는 것을 확인할 수 있다.

◆ 반면 Event-driven Approach는 하나의 스레드에서 모든 작업을 처리하기 때문에, 클라이언트 수가 많아질수록 상대적으로 동시 처리율이 낮아지는 경향을 보인다. 이는 이벤트 루프의 부담이 증가하면서 발생하는 오버헤드 때문이다.

◆ 클라이언트의 개수가 적을 때는 두 방식의 차이가 미세하지만, 클라이언트 수가 증가할수록 Thread-Based Approach의 성능이 더 우수해진다. 이는 Thread-Based Approach가 많은 클라이언트를 병렬적으로 처리하는 반면, Event-Based Approach는 순차적으로 처리하기 때문이다.

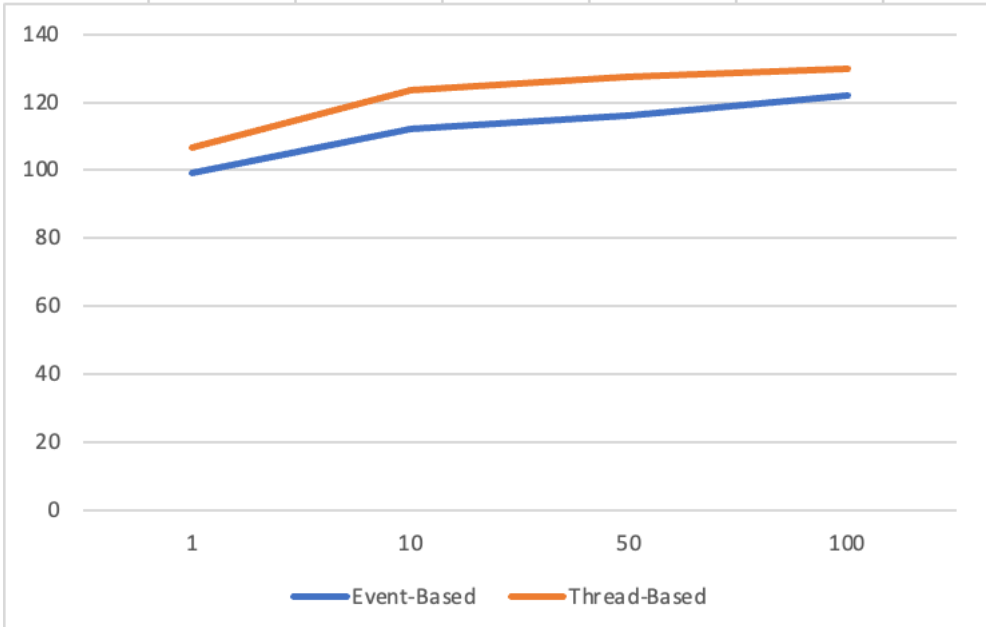
✓ 워크로드 분석

◆ 모든 클라이언트가 buy 또는 sell을 요청하는 경우

- Event-driven Approach – 1 client
  - `Total elapsed time for all clients: 0.102425 seconds`
  - `Total elapsed time for all clients: 0.100562 seconds`
  - `Total elapsed time for all clients: 0.099928 seconds`
- Event-driven Approach – 10 clients
  - `Total elapsed time for all clients: 0.863291 seconds`
  - `Total elapsed time for all clients: 0.935264 seconds`
  - `Total elapsed time for all clients: 0.876555 seconds`
- Event-driven Approach – 50 clients
  - `Total elapsed time for all clients: 4.458608 seconds`
  - `Total elapsed time for all clients: 3.785621 seconds`
  - `Total elapsed time for all clients: 4.655811 seconds`
- Event-driven Approach – 100 clients
  - `Total elapsed time for all clients: 7.518094 seconds`
  - `Total elapsed time for all clients: 8.904425 seconds`
  - `Total elapsed time for all clients: 8.165838 seconds`
- Thread-based Approach – 1 client
  - `Total elapsed time for all clients: 0.087775 seconds`
  - `Total elapsed time for all clients: 0.096125 seconds`
  - `Total elapsed time for all clients: 0.097737 seconds`
- Thread-based Approach – 10 clients
  - `Total elapsed time for all clients: 0.828485 seconds`
  - `Total elapsed time for all clients: 0.867278 seconds`
  - `Total elapsed time for all clients: 0.733556 seconds`
- Thread-based Approach – 50 clients
  - `Total elapsed time for all clients: 3.960295 seconds`

- `Total elapsed time for all clients: 3.770381 seconds`
- `Total elapsed time for all clients: 4.047595 seconds`
- Thread-based Approach – 100 clients
- `Total elapsed time for all clients: 7.678376 seconds`
- `Total elapsed time for all clients: 7.644438 seconds`
- `Total elapsed time for all clients: 7.795759 seconds`

Client (10 requests per client)		1	10	50	100
Event-Based	1st (s)	**0.102425**	**0.863291**	**4.458608**	**7.518094**
	2nd (s)	**0.100562**	**0.935264**	**3.785621**	**8.904425**
	3rd (s)	**0.099928**	**0.876555**	**4.655811**	**8.165838**
	Avg (s)	0.100971667	0.891703333	4.300013333	8.196119
	동시처리율	99.03768384	112.1449211	116.2787092	122.0089655
Thread-Based	1st (s)	**0.087775**	**0.828485**	**3.960295**	**7.678376**
	2nd (s)	**0.096125**	**0.867278**	3.770381	**7.644438**
	3rd (s)	**0.097737**	**0.733556**	**4.047595**	**7.795759**
	Avg (s)	0.093879	0.809773	3.926090333	7.706191
	동시처리율	106.520095	123.4913982	127.3531574	129.7657948



- buy와 sell 명령어는 주식 데이터를 수정해야 하기 때문에 동기화 오버헤드가 발생할 수 있다. 특히, Thread-Based Approach에서는 각 스레드가 주식 데이터를 병렬로 수정하려고 할 때 발생하는 동기화 문제로 인해 성능이 저하될 수 있다. 따라서 동시 처리율이 모든 명령어를 섞어서 요청할 때보다 낮다.

- Event-driven Approach: 하나의 스레드가 모든 작업을 처리하기 때문에, 동시 처리율이 상대적으로 낮다. 클라이언트 수가 증가함에 따라 동기화 오버헤드가 누적되어 성능이 저하될 가능성이 크다.
- Thread-Based Approach:
  - 클라이언트 수가 1일 때 동시 처리율은 106.52, 10일 때는 123.49, 50일 때는 127.35, 100일 때는 129.77로 증가한다.
  - 평균 응답 시간은 1명의 클라이언트가 요청할 때 0.093879초에서 100명의 클라이언트가 요청할 때 7.706191초로 증가했다.
  - 여러 스레드가 병렬로 작업을 처리하므로, 멀티코어 환경에서 더 높은 동시 처리율을 기대할 수 있다. 하지만 동기화 오버헤드 때문에 예상보다 높은 성능을 보이지 않을 수도 있다. 그래도 Event-driven Approach보다 높은 동시 처리율을 보였다.
- buy와 sell 명령어는 주식 데이터를 수정해야 하기 때문에, 이들 명령어만 요청하는 경우 동기화 오버헤드가 발생할 수 있다. 특히, Thread-Based Approach에서는 각 스레드가 주식 데이터를 병렬로 수정하려고 할 때 발생하는 동기화 문제로 인해 성능이 저하될 수 있다.
- Event-driven Approach는 하나의 스레드가 모든 작업을 처리하기 때문에, 동시 처리율이 상대적으로 낮다. 클라이언트 수가 증가함에 따라 동기화 오버헤드가 누적되어 성능이 저하될 가능성이 크다.
- Thread-Based Approach는 여러 스레드가 병렬로 작업을 처리하므로, 멀티코어 환경에서 더 높은 동시 처리율을 기대할 수 있다. 하지만 동기화 오버헤드 때문에 예상보다 높은 성능을 보이지 않을 수도 있다. 그래도 Event-driven Approach보다 높은 동시 처리율을 보였다.

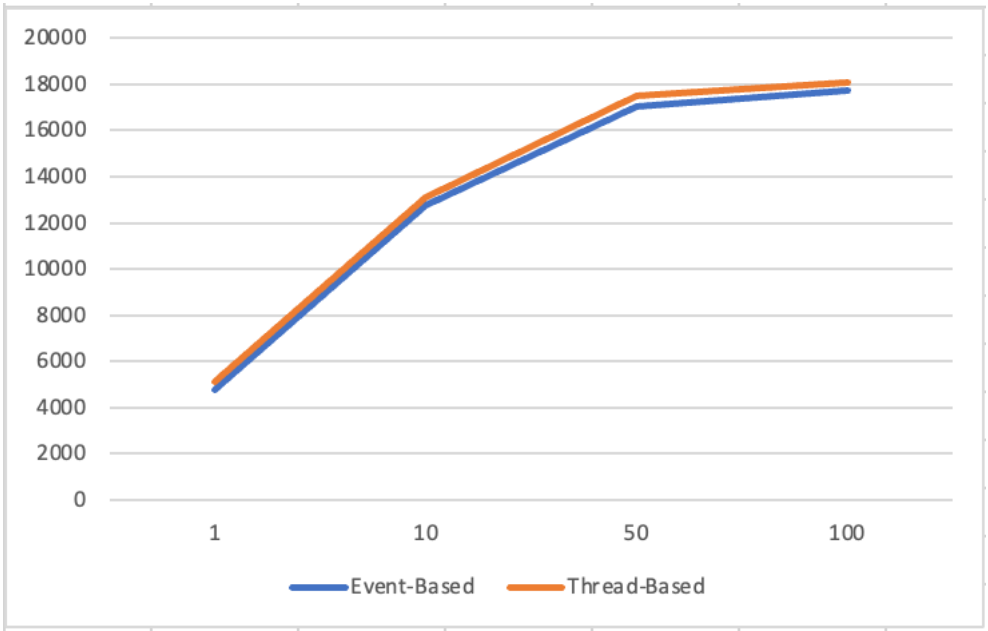
◆ 모든 클라이언트가 show만 요청하는 경우

- Event-driven Approach – 1 client
- `Total elapsed time for all clients: 0.002082 seconds`
- `Total elapsed time for all clients: 0.002246 seconds`
- `Total elapsed time for all clients: 0.001948 seconds`

- Event-driven Approach – 10 clients
  - `Total elapsed time for all clients: 0.008978 seconds`
  - `Total elapsed time for all clients: 0.007358 seconds`
  - `Total elapsed time for all clients: 0.007138 seconds`
- Event-driven Approach – 50 clients
  - `Total elapsed time for all clients: 0.030403 seconds`
  - `Total elapsed time for all clients: 0.027856 seconds`
  - `Total elapsed time for all clients: 0.029759 seconds`
- Event-driven Approach – 100 clients
  - `Total elapsed time for all clients: 0.055363 seconds`
  - `Total elapsed time for all clients: 0.055617 seconds`
  - `Total elapsed time for all clients: 0.058360 seconds`
- Thread-based Approach – 1 client
  - `Total elapsed time for all clients: 0.001994 seconds`
  - `Total elapsed time for all clients: 0.001962 seconds`
  - `Total elapsed time for all clients: 0.001932 seconds`
- Thread-based Approach – 10 clients
  - `Total elapsed time for all clients: 0.007372 seconds`
  - `Total elapsed time for all clients: 0.008118 seconds`
  - `Total elapsed time for all clients: 0.007363 seconds`
- Thread-based Approach – 50 clients
  - `Total elapsed time for all clients: 0.030273 seconds`
  - `Total elapsed time for all clients: 0.027298 seconds`
  - `Total elapsed time for all clients: 0.028107 seconds`
- Thread-based Approach – 100 clients
  - `Total elapsed time for all clients: 0.056314 seconds`

- `Total elapsed time for all clients: 0.054500 seconds`
- `Total elapsed time for all clients: 0.055190 seconds`

Client (10 requests per client)		1	10	50	100
Event-Based	1st (s)	**0.002082**	**0.008978**	**0.030403**	**0.055363**
	2nd (s)	**0.002246**	**0.007358**	**0.027856**	**0.055617**
	3rd (s)	**0.001948**	**0.007138**	**0.029759**	**0.05836**
	Avg (s)	0.002092	0.007824667	0.029339333	0.056446667
	동시처리율	4780.114723	12780.09713	17041.96869	17715.83796
Thread-Based	1st (s)	**0.001994**	**0.007372**	**0.030273**	**0.056314**
	2nd (s)	**0.001962**	**0.008118**	**0.027298**	**0.0545**
	3rd (s)	**0.001932**	**0.007363**	**0.028107**	**0.05519**
	Avg (s)	0.001962667	0.007617667	0.028559333	0.055334667
	동시처리율	5095.108696	13127.37934	17507.41147	18071.85369



- show 명령어는 주식 데이터를 읽기만 하므로 동기화 오버헤드가 발생하지 않는다. 따라서, show 명령어만 요청할 때 가장 높은 동시 처리율을 기대할 수 있다.
- Event-driven Approach:
  - 클라이언트 수가 1일 때 동시 처리율은 4780.11, 10일 때는 12780.10, 50일 때는 17041.97, 100일 때는 17715.84로 크게 증가했다.
  - 평균 응답 시간은 1명의 클라이언트가 요청할 때 0.002092초에서 100명의 클라이언트가 요청할 때 0.0564467초로 증가했다.

- 하나의 스레드가 모든 요청을 처리하지만, 읽기 작업만 수행하므로 동기화 오버헤드가 없고, 비교적 높은 동시 처리율을 유지할 수 있다.

- Thread-Based Approach:
  - 클라이언트 수가 1일 때 동시 처리율은 5095.11, 10일 때는 13127.38, 50일 때는 17507.41, 100일 때는 18071.85로 매우 크게 증가했다.
  - 평균 응답 시간은 1명의 클라이언트가 요청할 때 0.0019627초에서 100명의 클라이언트가 요청할 때 0.0553347초로 증가했다.
  - 각 스레드가 병렬로 읽기 작업을 수행하므로, 멀티코어 환경에서 매우 높은 동시 처리율을 보일 수 있다.
- show 명령어는 주식 데이터를 읽기만 하므로, 동기화 오버헤드가 발생하지 않는다. 따라서, show 명령어만 요청할 때 가장 높은 동시 처리율을 기대할 수 있다.
- Event-driven Approach는 하나의 스레드가 모든 요청을 처리하지만, 읽기 작업만 수행하므로 동기화 오버헤드가 없고, 비교적 높은 동시 처리율을 유지할 수 있다. 클라이언트 수가 증가함에 따라 동시 처리율이 증가한다.
- Thread-Based Approach는 각 스레드가 병렬로 읽기 작업을 수행하므로, 멀티코어 환경에서 매우 높은 동시 처리율을 보일 수 있다. 모든 클라이언트가 show 명령어만 요청하는 경우, Event-driven Approach보다 높은 동시 처리율을 보였다.

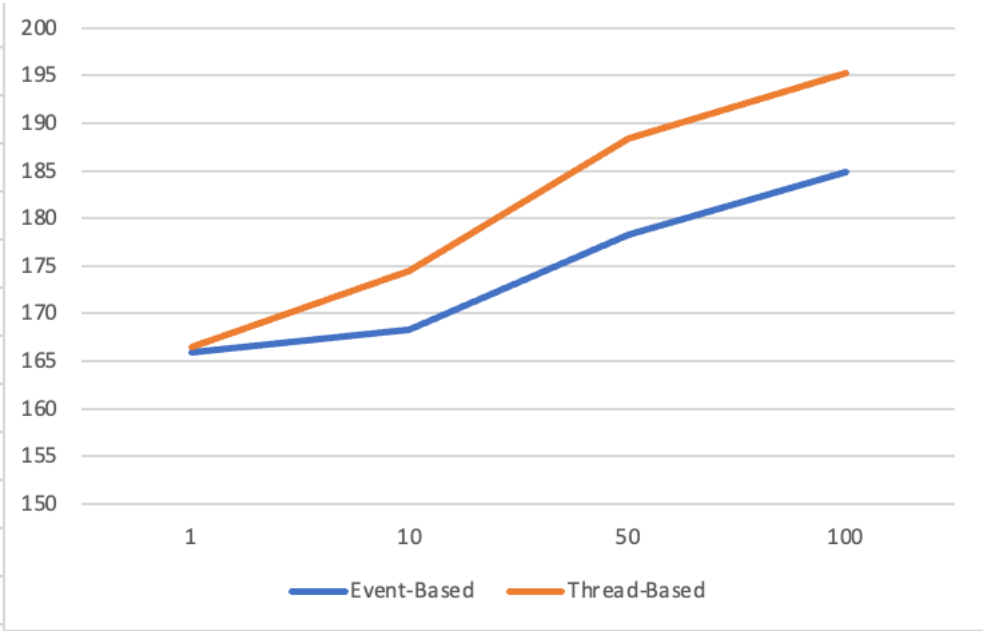
◆ Client가 buy, show 등을 섞어서 요청하는 경우

- Event-driven Approach – 1 client
- `Total elapsed time for all clients: 0.059565 seconds`
- `Total elapsed time for all clients: 0.054455 seconds`
- `Total elapsed time for all clients: 0.066790 seconds`
- Event-driven Approach – 10 clients

- `Total elapsed time for all clients: 0.599305 seconds`
- `Total elapsed time for all clients: 0.560385 seconds`
- `Total elapsed time for all clients: 0.622694 seconds`
- Event-driven Approach – 50 clients
- `Total elapsed time for all clients: 3.439288 seconds`
- `Total elapsed time for all clients: 2.511449 seconds`
- `Total elapsed time for all clients: 2.459569 seconds`
- Event-driven Approach – 100 clients
- `Total elapsed time for all clients: 5.340226 seconds`
- `Total elapsed time for all clients: 5.015325 seconds`
- `Total elapsed time for all clients: 5.875161 seconds`
- Thread-based Approach – 1 client
- `Total elapsed time for all clients: 0.048480 seconds`
- `Total elapsed time for all clients: 0.052351 seconds`
- `Total elapsed time for all clients: 0.079414 seconds`
- Thread-based Approach – 10 clients
- `Total elapsed time for all clients: 0.542926 seconds`
- `Total elapsed time for all clients: 0.653285 seconds`
- `Total elapsed time for all clients: 0.522852 seconds`
- Thread-based Approach – 50 clients
- `Total elapsed time for all clients: 2.854542 seconds`
- `Total elapsed time for all clients: 2.510160 seconds`
- `Total elapsed time for all clients: 2.598230 seconds`
- Thread-based Approach – 100 clients
- `Total elapsed time for all clients: 5.123416 seconds`
- `Total elapsed time for all clients: 5.036947 seconds`

- `Total elapsed time for all clients: 5.203286 seconds`

Client (10 requests per client)		1	10	50	100
Event-Based	1st (s)	0.059565	0.599305	3.439288	5.340226
	2nd (s)	0.054455	0.560385	2.511449	5.015325
	3rd (s)	0.06679	0.622694	2.459569	5.875161
	Avg (s)	0.06027	0.594128	2.803435333	5.410237333
	동시처리율	165.9200265	168.3138987	178.3526069	184.834775
Thread-Based	1st (s)	0.04848	0.542926	2.854542	5.123416
	2nd (s)	0.052351	0.653285	2.51016	5.036947
	3rd (s)	0.079414	0.522852	2.59823	5.203286
	Avg (s)	0.060081667	0.573021	2.654310667	5.121216333
	동시처리율	166.4401232	174.513674	188.3728255	195.2661116



- 다양한 명령어가 혼합되어 요청될 때, 각 명령어의 비용과 명령어 간의 상호작용이 성능에 영향을 미친다. 따라서, 동시 처리율은 클라이언트가 특정 명령어만 요청할 때보다는 낮다.
- Event-driven Approach:
  - 클라이언트 수가 1일 때 동시 처리율은 165.92, 10일 때는 168.31, 50일 때는 178.35, 100일 때는 184.83로 증가한다.
  - 평균 응답 시간은 1명의 클라이언트가 요청할 때 0.06027초에서 100명의 클라이언트가 요청할 때 5.410237초로 증가했다.
  - 하나의 스레드에서 모든 요청을 처리하므로, 혼합된 명령어를 처리할 때 발생하는 오버헤드로 인해 동시 처리율이 낮아질 수 있

다.

- Thread-Based Approach:
  - 클라이언트 수가 1일 때 동시 처리율은 166.44, 10일 때는 174.51, 50일 때는 188.37, 100일 때는 195.27로 증가한다.
  - 평균 응답 시간은 1명의 클라이언트가 요청할 때 0.0600817초에서 100명의 클라이언트가 요청할 때 5.1212163초로 증가했다.
  - 각 명령어가 별도의 스레드에서 병렬로 처리되므로, 전체적인 성능이 Event-driven Approach보다 우수하다. 하지만, show 명령어와 같은 높은 비용의 명령어가 포함되면 성능 저하가 발생할 수 있다.
- 다양한 명령어가 혼합되어 요청될 때, 각 명령어의 비용과 명령어 간의 상호작용이 성능에 영향을 미친다. 따라서, 동시 처리율은 클라이언트가 특정 명령어만 요청할 때보다는 낮아질 수 있다.
- Event-driven Approach는 하나의 스레드에서 모든 요청을 처리하므로, 혼합된 명령어를 처리할 때 발생하는 오버헤드로 인해 동시 처리율이 낮아질 수 있다. 특히, show 명령어와 같은 높은 비용의 명령어가 포함될 때 성능 저하가 발생할 수 있다.
- Thread-Based Approach는 각 명령어가 별도의 스레드에서 병렬로 처리되므로, 전체적인 성능이 Event-driven Approach보다 우수하다. 하지만, show 명령어와 같은 높은 비용의 명령어가 포함되면 성능 저하가 발생할 수 있다. 그럼에도 불구하고, Thread-Based Approach는 멀티코어 환경의 이점을 잘 활용하여 상대적으로 높은 동시 처리율을 유지한다.